FlashFire: Overcoming the Performance Bottleneck of Flash Storage Technology

Hyojun Kim and Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
{*hyojun.kim, rama*}*@cc.gatech.edu*

Abstract

Flash memory based Solid State Drives (SSDs) are becoming popular in the market place as a possible low-end alternative to hard disk drives (HDDs). However, SSDs have different performance characteristics compared to traditional HDDs, and there has been less consideration for SSD technology at the Operating System (OS) level. Consequently, platforms using SSDs are often showing performance problems especially with low-end SSDs.

In this paper, we first identify the inherent characteristics of SSD technology. Using this as the starting point, we propose solutions that are designed to leverage these characteristics and overcome the inherent performance problems of SSDs. At a macro-level, we propose a device driver-level solution called *FlashFire* that uses a *Cluster Buffer* and *Smart Scheduling* of read/write I/O requests from the OS. The net effect of this solution is to aggregate the small random writes from the OS into large sequential writes, and then sending them to the physical storage. We have implemented FlashFire in Windows XP and have conducted extensive experimental studies using disk benchmark programs as well as real workloads to validate its performance potential. We verified that FlashFire is able to provide better performance tuned to the intrinsic characteristics of SSD storages. For instance, the slowest netbook took 74 minutes to install MS Office 2007 package, and the time was reduced to 16 minutes with FlashFire. It is about 4.6 times better performance than before.

1 Introduction

NAND flash memory is enabling the rapid spread of SSD technology. Despite the fact that a magnetic disk is well entrenched in the storage market, an SSD is attractive for several reasons: it is small, light-weight, shock resistant, and energy efficient. These characteristics make SSDs attractive for mobile platforms, especially for laptops.

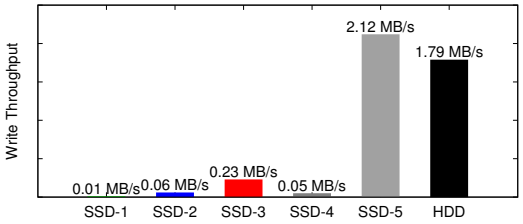

Figure 1: 4Kbytes write throughput measured using CrystalDiskMark benchmark.

In addition, the price of a SSD *scales down with size* much more gracefully than the price of a HDD does, thus smaller capacity SSDs are widely used for cheap *netbooks*. However, SSD-based netbooks show very poor I/O performance, especially for random writes compared to HDD-based systems. Figure 1 presents the results of our experiment with five laptops equipped with different SSDs and one with a HDD. We measure the throughput for randomly writing 4 Kbyte sized blocks to the storage system using CrystalDiskMark [9] benchmark. Except for one SSD (labeled SSD-5), the other four low-end SSDs (SSD-1 - SSD-4) show much lower throughputs compared to the HDD.

The reason for showing this figure is to underscore one of the inherent limitations of the SSD technology. NAND flash memory has different physical characteristics compared to magnetic storages. Due to the nature of the technology, NAND flash memory can be updated only in big chunks [18]. Thus, large sequential writes to the storage is not a problem with SSDs. However, small writes (i.e., random writes to the storage) result in poor performance as can be seen in Figure 1. High-end SSDs use additional resources (write buffer implemented using RAM and increased computing power) to compensate for this inherent limitation of the technology and achieve a modest increase in random write performance [5, 7].

The thesis of this paper is that a better design of the

lower levels of the OS software stack will overcome the performance limitation of the SSD technology. There have been some recent studies that have shown that software techniques can be successfully used to overcome the poor performance of a SSD for random writes.

FlashLite [14] proposes a user level library to convert random writes to sequential writes at the application level for P2P file-sharing programs. While this is good for specific applications, we believe that the problem should be tackled in the OS itself to make SSD-based storage a viable alternative to magnetic disk.

As is evident, most random writes stem from the well-known "small write" problem in file systems. Log-structured file system [22] has been proposed as a solution to the small write problem, and it is a promising approach for SSD-based file systems as well since it translates random writes to sequential writes. JFFS2 [21] and YAFFS [17] are well-known log-structured file systems working on Memory Technology Devices (MTDs), and NILFS [16] is for regular disks including HDDs and SSDs. However, due to the log-structured nature, such file systems have expensive garbage collection and scalability issues.

Similar to log-structured file system, EasyCo commercially provides block driver-level logging solution, called Managed Flash Technology (MFT) [6]. It has the nice property that it can be applied to existing file systems and OSes with minimal effort. However, it has the same issues as log-structured file system.

Solutions have been proposed to modify the OS-level cache management strategy for flash storages recognizing the relative expense of writes as opposed to reads. For example, Clean First Least Recently Used (CFLRU) [20] proposes a modification to the LRU strategy by skipping over dirty pages while selecting a victim for replacement. Although the CFLRU solution may reduce the amount of writes, it does not solve the random write problem, which is the focus of this paper.

This quick summary of the state-of-the-art summarizes various strategies that have been tried thus far to overcome the poor write performance of flash storage. The solution space spans all the way from application level down to the block device drive level. One part of the solution space that has not been investigated is at the level of the device level I/O scheduler [3, 11, 24]. Disk scheduling has a long history and some of the ideas therein have been incorporated in the Linux I/O scheduler that sits between the OS buffer cache and the physical storage. It keeps a request queue per storage device, and optimizes disk requests by rescheduling and merging requests. The basic idea is to minimize head movement by reordering the request queue and merging adjacent requests.

In this work, we propose a novel solution to combat the performance problem of flash-based storage system. Our system, called *FlashFire*, sits in between the OS I/O buffers and the physical device. Figure 2 shows the position of FlashFire in the software architecture of the storage system. The design of FlashFire is inspired by the I/O scheduler of Linux and the write buffer implemented using RAM that is inside high-end flash storage device itself.

The functionality of FlashFire can be summarized quite succinctly. It allocates a small portion of the host memory (32 Mbytes in the example implementation to be presented shortly) as a cluster buffer. Dynamically, the OS write requests to the flash storage (which may not necessarily be sequential) are converted to big sequential writes in the cluster buffer by FlashFire. Further, FlashFire dynamically reorders and merges OS write requests, respecting the physical characteristics of the flash storage. Thus, the OS write requests (random or not), stay in the cluster buffer and are flushed to the physical storage as large sequential writes. FlashFire may even read some sectors from the physical storage to "pad" the writes ensuring that they are large and sequential.

FlashFire plays three major roles. First, it absorbs small writes and emits big sequential writes to the physical storage. Second, it reduces the number of write requests between the host computer and the physical storage. Last, buffering in FlashFire ensures stable write response time regardless of the size of the write request. In other words, FlashFire serves a scheduling role by performing writes to the physical storage during idle times. High-end SSDs may have a write buffer inside the storage device. However, most netbooks generally use low-end SSDs that do not have a write buffer inside the storage device. In either case, FlashFire serves to reduce the number of write requests from the host to the physical device.

Reducing the number of write requests to the flash-based storage is very important. This is because write to the flash memory has to be preceded by a block erasure. As it turns out, a block can be erased only a finite number of times. High-end SSDs resort to wear leveling techniques in hardware to ensure that the writes are spread out over all the blocks of the flash memory chips. This, of course, has the downside of increasing write latency. Thus, an added bonus of FlashFire is a potential increase in the lifetime of the SSD-based storage.

The design of FlashFire allows it to be easily implemented at the device driver level. Further, this design choice allows integration of FlashFire in a system that supports both magnetic disk and SSD. We have implemented FlashFire as a driver level hook into Windows XP. We have evaluated FlashFire on four netbooks that use low-end SSDs, and one laptop computer that uses mid-level SSD. We executed three disk benchmark

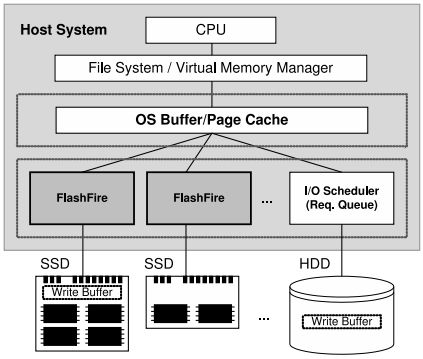

Figure 2: OS Software Stack Incorporating FlashFire

programs to measure write throughput. From all three benchmarks, we have verified that FlashFire provides substantial increase in throughput for small writes. For a more realistic evaluation, we tested several write intensive real workloads such as copying MP3 files and installing a huge software package. From the results, we show that FlashFire removes storage system bottleneck that is typical of SSD-based netbooks. Third, we have simulated the internals of SSD-storage and have recorded the number of block erasures incurred by FlashFire. The simulation result shows the considerable reduction in the number of block erasures and hence supports our hypothesis that FlashFire would increase the lifetime of SSD-based netbooks. Finally, we have carried out detailed studies on the effects of varying some of the design parameters of FlashFire on the performance.

We make the following contributions through this work. First we present a set of principles for combating the performance problems posed by the inherent characteristics of SSD technology. These principles include sector clustering, efficient LRU strategy for retiring clusters from the cluster buffer, cluster padding to reduce the number of block erasures, and deferred checking of the cluster buffer during read accesses. The second contribution is the FlashFire architecture itself that embodies these principles at the device driver level. FlashFire solves the (small and hence) random write problem of SSD without sacrificing read performance. Further, the design allows co-existence of SSD with other storage technologies without impacting their performance. The third contribution is a proof of concept implementation of FlashFire in Windows XP to validate the design ideas. The fourth contribution is a detailed experimental study to show that FlashFire does deliver on the promise of the proposed design. For a fifth and final contribution, we have shown through simulation that FlashFire offers the possibility of extending the lifetime of SSD-based storage system. We also opened FlashFire for public users, and have been receiving many positive feedbacks over six monthes.

The rest of the paper is organized as follows. Section 2 presents the background, specifically the state-of-the-art in SSD technology. Section 3 presents the principles underlying the FlashFire design and details of its implementation in Windows XP. We have conducted a detailed performance evaluation of FlashFire, which is described in Section 4. Section 5 presents concluding remarks and directions for future research.

2 Background

Before we present the design of FlashFire, it is useful to understand the state-of-the-art in Flash-based SSD storage technology. We also summarize another key technology, I/O scheduler in Linux, that serves as an inspiration for FlashFire design.

2.1 Flash Memory

Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before being written [18]. In flash memory, the amount of electric charges in a transistor represents 1 or 0. The charges can be moved both into a transistor by write operation and out by an erase operation. By design, the erase operation, which sets a storage cell to 1, works on a bigger number of storage cells at a time than the write operation. Thus, flash memory can be written or read a single page at a time, but it has to be erased in an erasable-block unit. An erasable-block consists of a certain number of pages. In NAND flash memory, a page is similar to a HDD sector, and its size is usually 2 Kbytes.

Flash memory also suffers from a limitation on the number of erase operations possible for each erasable block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. In single level cell (SLC) NAND flash memory, the expected number of erasures per block is 100,000 and this is reduced to 10,000 in two bits multi-level cell (MLC) NAND flash memory. If some erasable blocks that contain critical information are worn out, the whole memory becomes useless even though many serviceable blocks still exist. Therefore, many flash memory-based devices use wear-leveling techniques to ensure that erasable blocks wear out evenly [4].

2.2 Architecture of SSD

An SSD is simply a set of flash memory chips packaged together with additional circuitry and a special piece of software called Flash Translation Layer (FTL) [1, 10, 12, 19]. The additional circuitry may include a RAM buffer

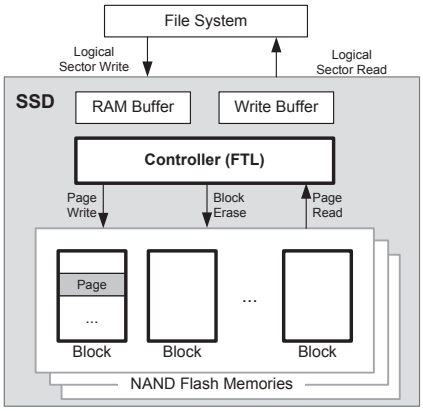

Figure 3: SSD, FTL and NAND flash memory. *FTL emulates sector read and write functionalities of a hard disk allowing conventional disk file systems to be implemented on NAND flash memory*

for storing meta-data associated with the internal organization of the SSD, and a write buffer for optimizing the performance of the SSD. The FTL provides an external logical interface to the file system. A sector[1] is the unit of logical access to the flash memory provided by this interface. A page inside the flash memory may contain several such logical sectors. The FTL maps this logical sector to physical locations within individual pages [1]. This interface allows FTL to emulate a HDD so far as the file system is concerned (Figure 3). To keep the discussion simple, we use sector and page interchangeably in this paper.

2.3 Characteristics of SSD

Agrawal et al. enumerate the design tradeoffs of SSDs in a systematic way, from which we can get a good intuition about the relation between the performance of SSD and the design decisions [1]. However, the fact of the matter is that without the exact details of the internal architecture of the SSD and the FTL algorithm, it is very difficult to fully understand the external characteristics of SSDs [5].

Nevertheless, at a macro level we can make two observations about SSD performance. First, they show their best performance for sequential read/write access patterns. Second, they show the worst performance for random write patterns.

At the device level, more complicated FTL mapping algorithms with more resources have been proposed to get better random write performance [19]. However, due to the increased resource usage of these approaches, they are used usually for high-end SSDs.

Incorporating a write-buffer inside the SSD is a slightly higher level approach than the FTL approach. For example, Kim and Ahn have proposed Block Padding Least Recently Used (BPLRU) [13] as a buffer management scheme for SSDs, and showed that even a small amount RAM-based write buffer could enhance random write performance of flash storage significantly.

2.4 I/O Scheduler in Linux

Another technology that serves as an inspiration for FlashFire design is the large body of work that exists in optimizing disk scheduling [11, 24]. Such optimizations have been embodied in the Linux I/O scheduler. Its primary goal is to reduce the overall seek time of requests, which is the dominant detriment to I/O performance on HDDs. I/O scheduler performs two main functions: *sorting* and *merging*. It keeps a list of pending I/O requests sorted by block number (i.e., a composite of cylinder, track, and sector number on the disk). A new request is inserted into this sorted list taking into account the block number of the new request. Further, if two requests in the sorted list are to adjacent disk blocks, then they are merged together. The sorting function ensures that the head movement is minimized. The merging function reduces the number of requests communicated from the host to the physical storage.

In the Linux architecture, each disk drive has its own I/O scheduler since the optimizations have to specific to the details of each individual disk drive. The OS has a unified buffer cache as shown in Figure 2.

3 FLASHFIRE

At a macro level it combines the functionalities of write buffer found in high-end SSDs, and the principles of I/O scheduler found in many OSes (e.g., Linux). In a nutshell, FlashFire is a device-driver level solution that uses a software write-buffer called *cluster buffer* as a staging area to aggregate sector writes from the OS, and schedules writes to the physical storage at opportune times. The ultimate goals are several fold: (a) reduce the number of I/O requests flowing to the physical device, (b) perform large sequential writes to the storage device to increase the performance, and (c) reduce the number of block erasures and thus increase the potential lifetime of the storage device. Essentially, FlashFire overcomes the performance problem of SSDs for dealing with random and small write patterns.

[1]Even though the term *sector* represents a physical block of data on a HDD, it is commonly used as an access unit for the FTL because it emulates a HDD. We adopt the same convention in this paper.

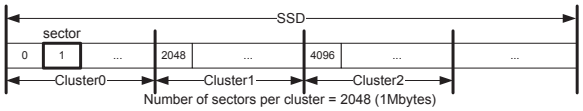

Figure 4: A cluster is defined as a set of sequential sectors

3.1 Sector, Cluster and Cluster-Based Victim Selection

The first principle deals with the *granularity* of write requests to the physical device. While a logical sector is the unit of read/write access presented by the SSD to the system software, such fine-grained access results in poor performance. Therefore, we define a *cluster* as the unit of I/O between the device driver and the SSD. A cluster is a fixed set of *sequential* sectors in the SSD. Figure 4 shows an example wherein a cluster is comprised of 2K sequential sectors. From the point of view of the device driver, the SSD is composed of a set of clusters. Thus, the unit of I/O between the device driver and the SSD is a "cluster" composed of some number of sequential sectors.

The optimal choice of the cluster size is a function of the intrinsic characteristics of each specific SSD. Ideally, the cluster size should be chosen to maximize the sequential write throughput for each specific SSD. This choice depends on both the internal hardware architecture of the SSD as well as the FTL mapping algorithm used inside the SSD. However, such information is not readily available to normal users. Besides, to accommodate generational changes in the SSD technology, the device driver should be designed to self-tune and figure out the optimal choice of cluster size.

One possible method to decide the cluster size is to treat the SSD as a black box, and observe the write throughput as a function of increasing cluster size. The optimal choice is one, for which the storage device yields the best write throughout. In general, through experimentation, we have determined that a bigger cluster size is always a safe choice for increasing the write throughput. However, a smaller cluster size is desirable for other considerations such as efficient usage of the cluster buffer and reliability in the presence of system crashes. In practice, we have found that it is not easy to determine the internal FTL algorithm using the black box approach. We could conclusively discover the FTL algorithm for only one experimental platform (out of the 5 we have used in our study). We are investigating automatic tools for inferring the optimal cluster size inspired by the gray-box approach proposed by Arpaci et al. [2]

In this paper, we have used a cluster size of 1 Mbyte for the one SSD, for which we could successfully conclude the internal FTL algorithm. For the other four platforms we have chosen to use a "large enough" cluster size, namely, 2 Mbytes.

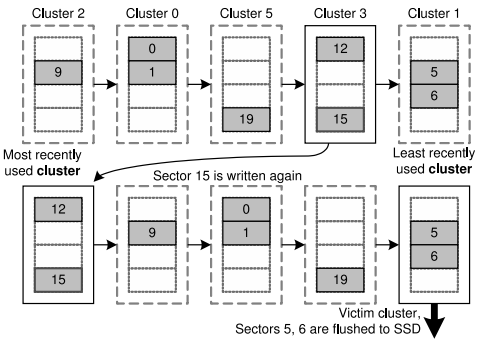

Figure 5: Victim selection in Cluster Buffer. *Each cluster can hold 4 contiguous logical sectors. Currently there are 8 logical sectors (0, 1, 5, 6, 9, 12, 15, and 19) present in the cluster buffer. Writing to cluster 15 results in the entire cluster being moved to the MRU position as shown. If a cluster has to be freed up, the LRU candidate is chosen and the logical sectors 5 and 6 will be flushed to the SSD.*

Another principle, closely allied to the choice of cluster size, is the free space management in the cluster buffer. Since cluster is the unit of I/O between the device driver and SSD, it is natural to use cluster as the unit of free space management. If a write request from the OS buffer cache targets a logical sector that is not currently in the cluster buffer, then space has to be allocated for the new sector. It is important to realize that the purpose of the cluster buffer is to aggregate writes before being sent to the storage device. It is not meant to serve as a cache for read accesses from the upper layers of the OS. Despite this intended use, an LRU policy for victim selection makes sense for the cluster buffer. The intuition is that if a cluster is not being actively written to by the OS, then it is likely that write activity for that cluster has ceased and hence can be retired to the physical device. However, victim selection is done at the cluster level rather than at the sector level. This is in keeping with the internal characteristics of SSDs. Recall that flash memory requires a block erasure to free up a physical block. Since a sector write to the physical storage could result in block erasures to free up space inside the flash memory, it makes sense to retire an entire cluster from the cluster buffer to amortize the potential cost of such block erasures. The cluster buffer is organized as an LRU list as shown in Figure 5. Upon a sector write that "hits" in the cluster buffer, the entire cluster containing that sector is moved to the MRU position of the list as shown in Figure 5. Victim selection uses an LRU policy, writing out all the sectors in the chosen victim cluster.

3.2 Early Retirement of Full Clusters

Using LRU for victim selection has a potential downside, especially when the total size of the cluster buffer is small. It should be emphasized that a small-sized cluster buffer suffices since the intended purpose is simply for aggregating sector writes. The phenomenon called *cache wiping* [25] results in most of the buffer space being used for housing sequentially written sectors of a large file.

The second principle that we propose avoids the pitfall of cache wiping and is called *early retirement of full clusters*. The idea is to move a fully written cluster to the tail of the LRU list to ensure that it will be chosen as a victim ahead of partially filled clusters and retired early from the cluster buffer.

3.3 Cluster Padding Using Block Read

Cluster-based LRU chooses a victim cluster and flushes all the written sectors in it to the physical storage. If there are a few non-contiguous sectors to be written out from a cluster, it increases the number of I/O operations from the device driver to the SSD. Decreasing the number of I/O operations is a key to achieving good overall performance of SSDs. For this reason, we propose another simple principle called *cluster padding*. The idea is to read the missing sectors from the device and write a full cluster back out to the device. At first glance, this may seem inefficient. However, since SSDs have very good read performance, the ability to perform one big sequential cluster write far outweighs the additional overhead of reading a few sectors for cluster padding. One has to be careful to ensure that the total number of I/O operations is not increased by cluster padding. For example, if there are a number of holes in the victim cluster, then several non-contiguous read operations may be needed to plug them, leading to inefficiency. The solution we propose is a novel *block read* for cluster padding. The idea is to issue a single block read request to the SSD, whose range extends from the smallest to the largest missing sector in the cluster. The valid sectors in the victim cluster overwrites the corresponding ones in the block read to create a full cluster, which can then be written out as one sequential write to the SSD. Essentially, this principle ensures that each eviction from the cluster buffer entails *at most* 2 I/O operations to the SSD. Figure 6 illustrates this block read cluster padding principle,

3.4 Deferred Checking of Buffer

This principle is to ensure the integrity of the data delivered to the OS buffer cache on a read request. With the cluster buffer in the mix, the device driver has to ensure that the correct data is delivered on a read request, if necessary getting it from the cluster buffer instead of the

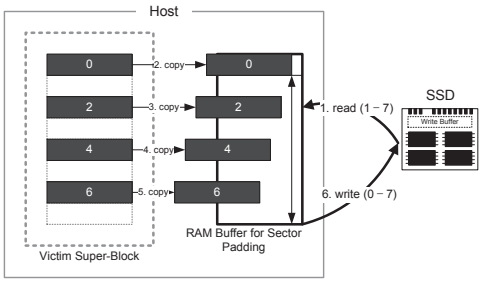

Figure 6: Block Read Cluster Padding.*Sectors 0, 2, 4, and 6 are the valid sectors in the victim cluster. A single block read of sectors 1-7 brings in the missing range of sectors from the SSD. The valid sectors in the victim cluster over-writes the corresponding ones fetched from the SSD into a temporary buffer. The merged buffer is written out to the SSD as one sequential write of sectors 0-7.*

physical storage. Figure 7 (a) shows a straightforward naive approach to handling the data integrity problem. Consider reading N sectors sequentially starting from some Logical Sector Number (LSN). The device driver checks whether a given sector is present in the cluster buffer or not. If a required sector is in the cluster buffer, the data is retrieved from it. Otherwise, the sector is read from the physical storage.

However, this naive approach of checking the cluster buffer has the same disadvantage as we discussed with cluster padding in the previous subsection. A single read request to a sequential set of clusters will get subdivided into several individual sector read requests. This is detrimental to the overall performance of the system due to the increased communication between the host and the physical storage.

We propose a principle called *deferred checking* of the cluster buffer to overcome this problem that can be detrimental to the read performance of SSD in the presence of the cluster buffer. The idea is to bypass the cluster buffer and send the read request directly to the SSD. Upon getting the sectors back from the SSD, the device driver checks the cluster buffer for the requested sectors, and replaces the sectors read from the SSD with the latest data from the cluster buffer. Figure 7 shows the flowchart embodying this principle.

Deferred checking is biased towards assuming that either the sectors are not in the cluster buffer, or that the cluster buffer contains a sparse number of non-contiguous sectors. Clearly, this scheme may result in communicating an unnecessary read request to the SSD if the entire set of sequential cluster is already present in the cluster buffer. However, there are several reasons why this may not be the common case. Firstly, the cluster replacement policy (see Section 3.2) favors early eviction

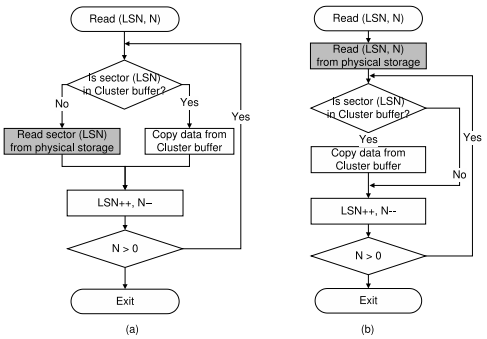

Figure 7: Flowchart of (a) native checking and (b) deferred checking of cluster buffer.*In deferred checking, only 1 I/O read request sent to the SSD. Cluster buffer is checked after receiving the data from the SSD.*

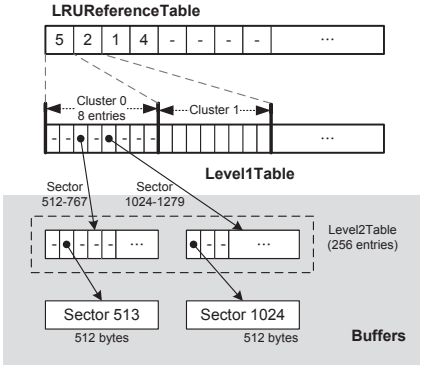

Figure 8: FlashFire DataStructures. *The value in the LRUReferenceTable indexed by the logical cluster number represents the timestamp of access to that logical cluster. LRU candidate is the cluster with the smallest timestamp.*

of such full clusters. Secondly, there is a higher likelihood that anything that is still in the cluster buffer is likely in the OS buffer/cache as well. By the same token, if some sectors that were recently written to are not in the OS buffer/cache, then it is highly unlikely that they are still in the cluster buffer either. In other words, the deferred checking is simply to serve as a safety net for sectors not yet retired from the cluster buffer. Nevertheless, we do plan to investigate other optimizations as part of future research to reduce unnecessary read requests to the physical storage if the sectors are already present in the cluster buffer.

3.5 Implementation

We have implemented FlashFire as a device driver in Windows XP. Being in the critical path of system I/O, it is extremely important that we use efficient data structures in FlashFire. Looking up the cluster buffer is an operation that occurs on every read/write access from the system. Hence, the data structures associated with this lookup is critical to overall performance. We use a simple two-level lookup scheme for locating a sector in the cluster buffer.

FlashFire has four major data structures: *LRUReferenceTable*, *Level1Table*, *Level2Table*, and *buffers*. Figure 8 shows the relationship between these data structures. In the figure, each sector is 512 bytes and the cluster size is assumed to be 1Mbytes. This two-level indexing scheme allows accessing any of the 2048 sector buffers corresponding to a logical cluster in constant time.

If there are no more buffers available in the buffer pool, FlashFire may have to evict a cluster to make room for a write request to a new sector. However, cluster eviction is not a common occurrence since the purpose of the cluster buffer is to aggregate and retire the sector writes to the physical storage in a timely manner. On the other hand, updating the LRU information is in the critical path of every sector access. For this reason, we use a statically allocated table (LRUReferenceTable) instead of a linked list for LRU management. The LRUReferenceTable has as many entries as the number of logical clusters in the physical storage. The value in the LRUReferenceTable indexed by the logical cluster number represents the timestamp of access to that logical cluster. A global 32-bit counter in FlashFire is used as the timestamp. This counter is incremented on each sector write, and the new value is written into the LRUReferenceTable indexed by the logical cluster number. Thus, updating the LRUReferenceTable takes constant time. To choose a victim, the logical cluster with the smallest timestamp has to be determined. We search the LRUReferenceTable to determine the victim cluster. It should be emphasized that updating the LRUReferenceTable is the common case and choosing a victim for eviction is not that common, since most likely we will always find a free sector buffer from the buffer pool. Hence, this design optimizes the common case.

4 Evaluation

We have conducted detailed experimental studies on five different state-of-the-art SSD-equipped platforms to understand the performance potential of FlashFire compared to native device drivers on Windows XP. Our studies include (a) throughput comparison using disk benchmarks such as ATTO benchmark, HDBENCH, and CrystalDiskMark, (b) elapsed time comparison for real workloads including copying MP3 files, extraction of zipped Linux sources, installing and uninstalling Microsoft Office, and web browsing, and (c) block erasure count com-

Table 1: Experimental Platforms.*First four are netbooks and the last one is a regular laptop.*

Disk	Host System Model	Size
SSD-1	Acer Aspire One AOA110-1626	16GB
SSD-2	Asus Eee PC1000-BK003	8GB
SSD-3	Dell Inspiron Mini 9	16GB
SSD-4	HP 1010NR	8GB
SSD-5	Dell XPS 1210 / OCZSSD2-1C128G	128GB

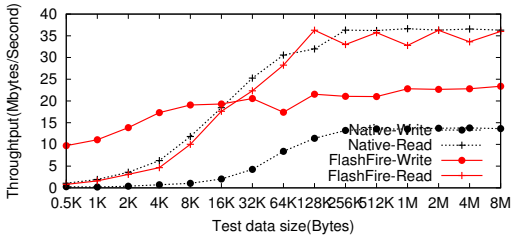

Figure 9: ATTO benchmark results on SSD-1.*Performance gain for FlashFire even for large (sequential) writes.*

parison (via simulation) as a measure of durability and lifetime of the SSD. Additionally, we have carried out detailed studies on the effects of varying some of the design parameters of FlashFire on the performance. In all of these studies our focus has been on measuring the effectiveness of FlashFire in meeting its stated design goals, namely, improving the write I/O performance of SSD, and reducing the block erasure count to extend the SSD lifetime. Thus, we have consciously chosen write intensive workloads for our experimental study.

4.1 Experimental Environment

Table 1 summarizes the hardware specification of the five platforms used in our study. Four of them are netbooks using Intel ATOM N270 processor, and the fifth is a regular laptop using Intel Core Duo processor. All five platforms have 1 Gbytes of system memory, and are configured with 32 Mbytes of Cluster Buffer in their respective FlashFire drivers. All five platforms run Windows XP Professional SP3 English edition on SSDs formatted with NTFS file system. To repeat the experiments without disk aging effect, disk imaging tool (Norton Ghost) has been used before each experiment.

4.2 Disk Benchmark Results

4.2.1 ATTO benchmark

This benchmark program first creates a 256 Mbyte master file. It then measures the throughput for reading and

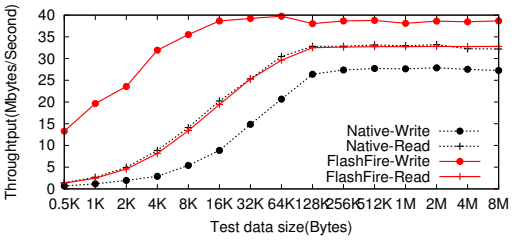

Figure 10: ATTO benchmark results on SSD-2.*Performance gain for FlashFire even for large (sequential) writes*

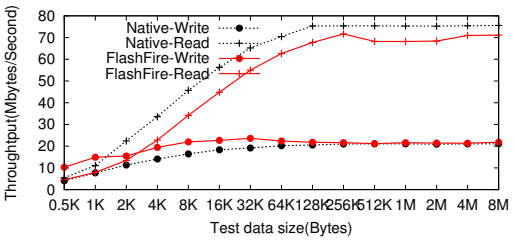

Figure 11: ATTO benchmark results on SSD-3.*Performance gain for FlashFire only for small (random) writes.*

writing the master file randomly with various granularity test sizes ranging from 512 bytes to 8 Mbytes. Thus, the generated workload is a mix of small (random) and large sequential accesses to the storage system. It should be noted that even for large write patterns, the workload generates additional small random writes (for file system meta-data) since the benchmark works through the file system.

Figures 9 - 13 show the results measured on the five platforms. The figures show the read/write throughputs measured with ATTO benchmark for the native device driver and FlashFire on each platform as a function of the test data sizes. The reported results are the averages of three repeated runs of the benchmark programs. In all five platforms, FlashFire incurs slight performance losses

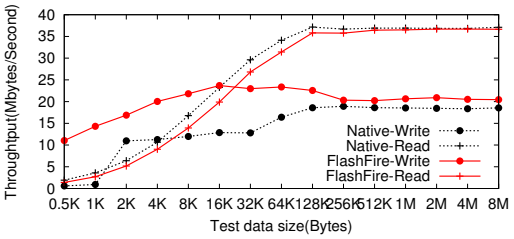

Figure 12: ATTO benchmark results on SSD-4.*Performance gain for FlashFire only for small (random) writes.*

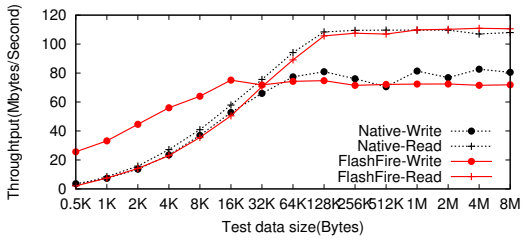

Figure 13: ATTO benchmark results on SSD-5.*Performance gain for FlashFire only for small (random) writes.*

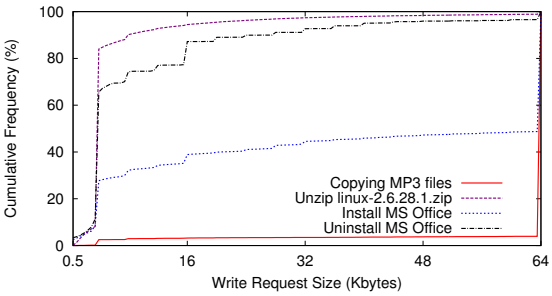

Figure 14: Cumulative frequencies of write request sizes for the four write-intensive workloads.

for reads, but shows significant performance gains especially for small writes. For example, on SSD-5 (a mid-level caliber SSD), FlashFire write throughput shows an improvement of roughly 73% compared to the native driver for 8 Kbyte data size.

4.2.2 Other Benchmarks

In addition to ATTO benchmark, we also experimented with two other disk benchmark programs: HDBENCH [8] and CrystalDiskMark [9]. Due to space limitations we do not present results of those experiments. The trends observed with these benchmark programs are similar to what we have reported with ATTO. In fact, the performance gain of FlashFire for sequential writes was even more noticeable with these benchmark program results. Since, we do not know the inner details of the benchmark programs, we can only guess that perhaps these programs use smaller test data sizes than ATTO for their sequential write patterns.

4.3 Real Workload Tests

The next set of experiments is designed to evaluate the performance gain of FlashFire while running real workloads. We have chosen five different workloads, four of which are write intensive, and the fifth has a mix of read and write traffic: (1) MP3 file copy, (2) File extraction

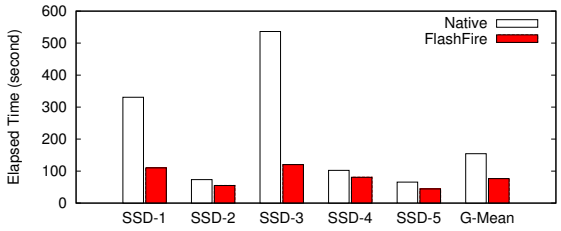

Figure 15: MP3 file copying results.*Geometric mean of elapsed time for FlashFire roughly half of the native driver.*

from compressed Linux sources, (3) Installation of Microsoft Office, (4) Un-installation of Microsoft Office, and (5) Web browsing 30 sites.

Figure 14 shows the cumulative frequencies of write request sizes for the four write-intensive workloads. We do not show the frequencies for the web browser workload because it is not reproducible due to network and server status. As can be seen from Figure 14, MP3 file copying is mostly composed of large writes, while unzipping Linux sources has small writes. MS Office installation is a mix of both big and small writes, and the un-installation may be classified as a small write workload.

We included web browsing as one of the workloads simply to get a feel for how FlashFire would perform for an average user activity, since it would generate both read and write traffic to the storage system. The last web browsing can show the overall performance effect of FlashFire including disk reads.

4.3.1 MP3 File Copy

The experiment involves copying copied total of 806 Mbytes of 102 MP3 files from an external USB HDD to the target SSD drive, and measuring the elapsed time. After the copy is complete, we delete the copied files and repeat the test three times and report an average value.

This workload mainly consists of large writes. Referring to Figure 14, the frequency for 64Kbyte sized writes is over 96%.

Figure 15 shows the results of this experiment on the five platforms. In all cases, FlashFire significantly reduces the elapsed time compared to the native driver. This result validates our earlier conjecture with the ATTO benchmark that the file system meta-data traffic (which is usually small writes) hurts the performance of SSDs even when the workload is mostly large writes. FlashFire alleviates this problem. The geometric mean of elapsed time is roughly 50% compared to the native driver on these platforms.

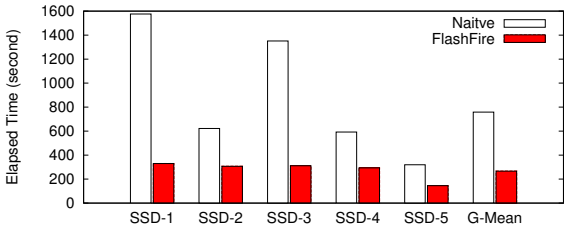

Figure 16: Linux source extraction test results on SSDs.*Geometric mean of elapsed time for FlashFire roughly 35% of the native drive*

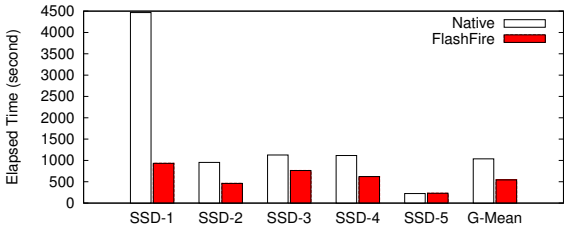

Figure 17: Microsoft Office install time.*Geometric mean of elapsed time for FlashFire roughly 52% of the native driver.*

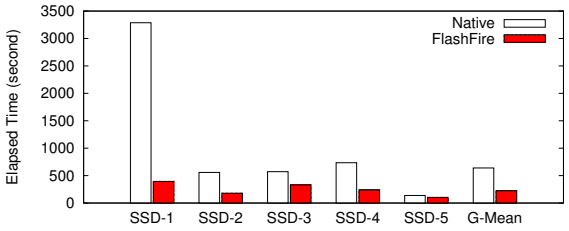

Figure 18: Microsoft Office uninstall time. *Geometric mean of elapsed time for FlashFire roughly 35% of the native driver.*

4.3.2 Decompress Linux Sources

This experiment is designed to verify the effects of FlashFire for a workload dominated by small writes. To prepare for the experiment, we extracted linux-2.6.28.1.tar.bz2 and recompressed it to a zip file since Windows XP compressed folder can only recognize zip format. Then, we extracted linux-2.6.18.1.zip from an external USB drive to the target SSD, and measured the elapsed time. After the extraction is over, we delete the extracted files and repeat the test three times to get an average value. In the workload, over 90% of the requests are smaller than 8Kbytes.

Figure 16 shows highly improved performance for FlashFire as expected, since the workload is dominated by small writes. SSD-3 using native driver, once again shows a behavior that is inconsistent with its benchmark performance for small write throughput. With FlashFire, the geometric mean of elapsed time is reduced from 758.3 seconds to 266.4 seconds (35% compared to the native driver)

4.3.3 Installing and Uninstalling Microsoft Office

The next experiment measures the elapsed time for installing and uninstalling a huge software package, Microsoft Office 2007. The install workload is well mixed with both big writes and small writes. Referring to Figure 18, about 51% of write requests are 64Kbyte sized, and 28% of the requests are smaller than 4Kbyte sized.

After the installation, we uninstall the package and measure the elapsed time once again. The uninstall workload is mainly composed of small writes. Like the other experiments, we repeat installing and uninstalling three times. By this repetition, we could also see the disk fragmentation effects with FlashFire.

Figures 17 and 18 show the averages of both install and uninstall times, respectively. The four low-end SSDs (SSD-1 to SSD-4) show much enhanced performance with FlashFire, but there is a slight performance loss in the case of the mid-level SSD (SSD-5) for install, and a modest gain for uninstall. Geometric means are reduced to 52% and 35% compared to the native driver for install and uninstall, respectively.

4.3.4 Web Browsing

When we browse web sites, a web browser downloads many small image files from its web server to a temporary directory in a local disk drive, and reads the files to render web pages. Therefore, both read and write throughput of the storage device is likely to have an influence on the speed of web browsing. Further, since low-end SSDs are commonly used for netbooks, this workload is an interesting one to study for understanding the performance potential of FlashFire.

For this purpose, we have designed and used a custom benchmark program named, *IEBench*. It is developed with Microsoft Visual C++ 6.0 and Internet browser control, which is the engine of Microsoft Internet Explorer. It visits 30 web sites continuously, and reports the total elapsed time.

By its nature, we cannot avoid the influence of external factors such as networking speed and the state of the web server that we visit. To minimize interferences, we repeated our experiments five times during mid-night (EST in the U.S.) with an internal proxy-server. The popular and stable 30 web sites in the United States are chosen for the test, and we cleared the Internet Explorer temporary directory before each test to see the disk related

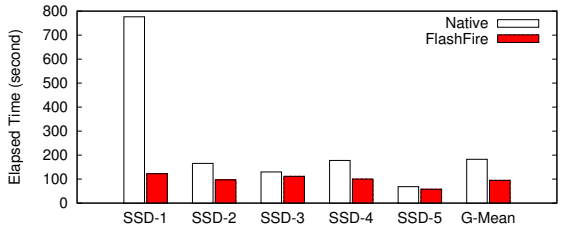

Figure 19: IEBench: Web browsing time for 30 US web sites. *Geometric mean of elapsed time for FlashFire roughly 52% of the native driver.*

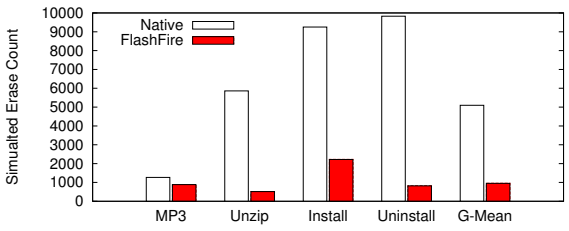

Figure 20: Simulated erase counts with collected write traces. *Geometric mean of erase count for FlashFire roughly 18% of the native driver.*

effects.

Figure 19 shows the results for the five SSDs, and as can be seen FlashFire improves Internet browsing performance significantly compared to the native driver on these devices. Geometric mean is reduced to 52% compared to the native driver using FlashFire.

4.4 Erase Count Simulation

Due to the nature of NAND flash memory, a block can be erased only a finite number of times. Therefore, one metric to evaluate the lifetime of an SSD is the cumulative count of block erasures. However, there is no known way to find the exact physical erase counts from an SSD.

To evaluate the durability effect of FlashFire, we use a simulation method. By assuming that the SSD is using log-block FTL [15] as its management algorithm, we have simulated the internal operations of the SSD to record the number of block erasures experienced inside the SSD. As inputs for the simulations, we used write traces collected from two different configurations with disk trace collecting tool, *Diskmon* [23]: (1) Original system and (2) FlashFire applied system. The input to the simulator are disk write traces collected using Diskmon for the five workload programs used in the performance evaluation using the native driver and FlashFire.

Figure 20 shows the simulated erase counts, and we can see that FlashFire is effectively reducing the number of erase counts. This means that FlashFire is beneficial not only for enhancing the performance of SSD but also for the durability of SSD. The geometric mean of erase counts for the four workloads is reduced from 5097 to 955 (FlashFire is 18% compared to the native driver).

4.5 Detailed Analysis

We have conducted a number of other experiments with FlashFire aimed to get a better understanding of its performance potential. For this set of experiments, we choose one target platform, namely, SSD-2, which showed significant gains for FlashFire. The tests we conduct are the following: analysis of write traces of Microsoft Office install workload; sensitivity to the Cluster Buffer size; and sensitivity to the file system.

Table 2: Write requests distribution for Microsoft Office installation using the native driver.

Req. Size	**Frequency (%)**		**Written Sectors (%)**	
128	17,974	51.3	2,030,672	85.4
8	6,716	19.2	53,728	2.0
32	1,305	3.7	41,760	1.6
1	884	2.5	884	0.03
7	815	2.3	5,705	0.2
16	765	2.2	12,240	0.5
56	466	1.3	26,096	1.0
64	413	1.2	26,432	1.0
3	311	0.9	933	0.03
24	304	0.9	7,296	0.3
Others	5,102	14.6	217,576	8.1
Total	35,055	100	2,693,322	100

4.5.1 Write Trace Analysis

The purpose of this test is to verify the effectiveness of FlashFire to aggregate small writes into large sequential writes. We analyzed collected write traces while installing Microsoft Office both with the native device driver and with FlashFire.

Table 2 shows the request size distribution of collected write traces while installing Microsoft Office using the native driver. We can see that there are various sizes of write requests ranging from 0.5 Kbytes to maximum 64 Kbytes.

Table 3: Write requests distribution for Microsoft Office installation using FlashFire.

Req. Size	**Frequency (%)**		**Written Sectors (%)**	
2048	2,222	100	4,550,656	100
Total	2,222	100	4,550,656	100

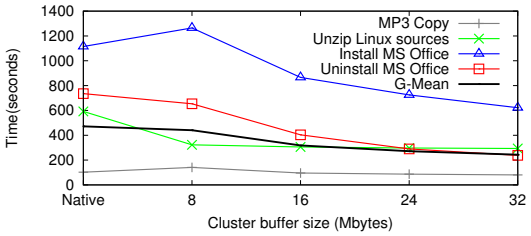

Figure 21: Effect of Varying the size of the Cluster Buffer. *Too small a Cluster Buffer (less than 16 Mbytes) can be detrimental to performance of FlashFire.*

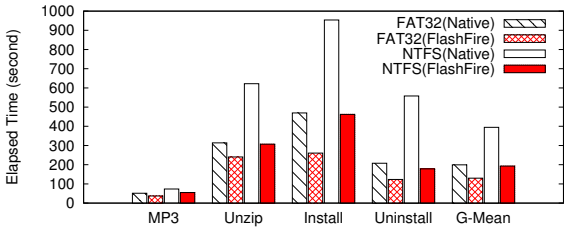

Figure 22: Sensitivity of FlashFire to File Systems. *FlashFire shows bigger performance gains with NTFS than FAT32.*

As can be seen from Table 3, FlashFire by design successfully converts all the requests to a fixed size of 1 Mbytes. It is interesting to note the total amount of sectors written in the two cases. FlashFire almost doubles the number of sectors written compared to the native driver. This is because of the cluster padding technique used in FlashFire to convert random writes to sequential writes. Despite this increase in the number of sectors written, the elapsed time for this workload with FlashFire is much smaller than with the native driver as we have already seen (543 seconds as opposed to 1037 seconds, as shown in Figure 17, G-Mean). Further, FlashFire also reduces the block erasures for this workload as already noted (2222 as opposed to 9282, Figure 20).

4.5.2 Sensitivity to Cluster Buffer Size

To show the effects of varying the size of the Cluster Buffer in FlashFire, we repeated our real workload tests with three more configurations in addition to the default 32 Mbytes size: 8 Mbytes, 16 Mbytes, and 24 Mbytes.

Figure 21 shows the results of this experiment. From the graph, we find that the cluster buffer that is too small in size may cause performance loss. When cluster buffer size is 8Mbytes, Microsoft Office install and MP3 copy workloads incur about 13% and 38% performance losses with FlashFire, respectively.

4.5.3 Sensitivity to File Systems

This experiment is intended to study the sensitivity of FlashFire to the two file systems available in Windows XP, namely, NTFS and FAT32. The default file system used in all the experiments thus far is NTFS. Figure 22 shows the elapsed time result using NTFS and FAT32 for the four workloads. On FAT32 file system, FlashFire shows smaller performance gains than on the NTFS file system. This is because NTFS file system generates more random writes than FAT32 file system in general.

5 Concluding Remarks

Flash memory based SSD storage is attractive in many ways. However, there are some inherent performance problems of SSD storage systems due to the nature of this technology. We have identified the specific problems that this technology has, and proved that a better design of the lower levels of the OS software stack can effectively solve the problems.

We opened FlashFire Windows XP version for public users on our project homepage, and its usefulness has been proven by a number of users especially by mid-level and low-end SSD owners. FlashFire seems to be not much beneficial for some high-end SSDs. Even though there is no way to find out exactly what techniques are used inside the high-end SSDs, the techniques are possibly equivalent to our FlashFire. We believe that our host-side approach is more general, economical, and power-efficient compared to the device-level approach.

If we compare FlashFire to higher level software solutions like SSD-aware file system, FlashFire approach is relatively simple, and can easily be applied to the existing systems without major modifications. As it is transparent to the existing storage components, users can freely control whether to use it or not at any time. In addition, our solution is orthogonal to the other software solutions such as log-structured file systems [16], and SSD-aware cache management schemes [20].

However, FlashFire brings one concern with reliability. Because it holds written sectors in the cluster buffer, a sudden power failure can hurt filesystem integrity. It may not possible to perfectly clear the issue due to the buffering nature, but we use several techniques to minimize the problem. FlashFire tries to flush its buffer as often as possible when the storage system is idle, and provides an easy interface to control its function dynamically. Users can disable FlashFire whenever they need to do mission critical works. Another possible solution is distinguishing important writes from normal writes, and let the important writes to bypass the cluster buffer.

While we have focused on SSDs in this study, we

would like to investigate the design of FlashFire to support removable flash memory storage. Further, since the overall tenet of FlashFire is to reduce the amount of I/O requests to the storage system, we would like to study its efficacy for other storage devices, especially HDDs.

References

[1] AGRAWAL, N., PRABHAKARAN, V., WOBBER, T., DAVIS, J. D., MANASSE, M., AND PANIGRAHY, R. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference* (Berkeley, CA, USA, 2008), USENIX Association, pp. 57–70.

[2] ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Information and control in gray-box systems. *SIGOPS Oper. Syst. Rev. 35*, 5 (2001), 43–56.

[3] BOVET, D., AND CESATI, M. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[4] CHANG, L.-P. On efficient wear leveling for large-scale flash-memory storage systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing* (New York, NY, USA, 2007), ACM, pp. 1126–1130.

[5] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (New York, NY, USA, 2009), ACM, pp. 181–192.

[6] COMPANY, E. C. Managed Flash Technology. `http://www.easyco.com/mft/index.htm`.

[7] DUMITRU, D. Understanding Flash SSD Performance. Draft, `http://www.storagesearch.com/easyco-flashperformance-art.pdf`, 2007.

[8] HDBENCH.NET. HDBENCH V3.40beta6. `http://www.hdbench.net/ja/hdbench/index.html`.

[9] HIYOHIYO. CrystalDiskMark. `http://crystalmark.info/software/CrystalDiskMark/index-e.html`.

[10] INTEL CORPORATION. Understanding the Flash Translation Layer (FTL) Specification. White Paper, `http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf`, 1998.

[11] IYER, S., AND DRUSCHEL, P. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM, pp. 117–130.

[12] KAWAGUCHI, A., NISHIOKA, S., AND MOTODA, H. A flash-memory based file system. In *USENIX Winter* (1995), pp. 155–164.

[13] KIM, H., AND AHN, S. BPLRU: a buffer management scheme for improving random writes in flash storage. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2008), USENIX Association, pp. 1–14.

[14] KIM, H., AND RAMACHANDRAN, K. Flashlite: A user-level library to enhance durability of ssd for p2p file sharing. In *ICDCS '09: Proceedings of the 2008 The 29th International Conference on Distributed Computing Systems* (2009).

[15] KIM, J., KIM, J. M., NOH, S., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for CompactFlash Systems. *IEEE Transactions on Consumer Electronics 48*, 2 (2002), 366–375.

[16] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The linux implementation of a log-structured file system. *SIGOPS Oper. Syst. Rev. 40*, 3 (2006), 102–107.

[17] LTD, A. O. Yaffs: A NAND-Flash Filesystem. `http://www.yaffs.net`.

[18] M-SYSTEMS. Two Technologies Compared: NOR vs. NAND. White Paper, `http://www.dataio.com/pdf/NAND/MSystems/MSystems\_NOR\_vs\_NAND.pdf`, 2003.

[19] PARK, C., CHEON, W., KANG, J., ROH, K., CHO, W., AND KIM, J.-S. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *Trans. on Embedded Computing Sys. 7*, 4 (2008), 1–23.

[20] PARK, S.-Y., JUNG, D., KANG, J.-U., KIM, J.-S., AND LEE, J. CFLRU: a replacement algorithm for flash memory. In *CASES '06: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (New York, NY, USA, 2006), ACM, pp. 234–241.

[21] REDHAT. JFFS2: The Journalling Flash File System, version 2. `http://sources.redhat.com/jffs2`.

[22] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[23] RUSSINOVICH, M. DiskMon for Windows v2.01. `http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx`, 2006.

[24] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk scheduling revisited. pp. 313–324.

[25] VAHALIA, U. *UNIX internals: the new frontiers*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1996.